下载APP

06 | TreeMap：红黑树真的有那么难吗？

2021-12-23 黄清昊

《算法实战高手课》 课程介绍 >

讲述：黄清昊

时长 20:34 大小 18.85M

你好，我是微扰君。

上一讲，我们讲到如何利用散列表解决类似“文档中不同单词计数”的问题，并以 JDK 中 HashMap 的实现为例讲解了散列表背后的思想。

单词计数这个问题最基本的解决思路就是建立一个线性的符号表，每次计数的时候，遍历符号表就可以找到对应单词的计数器，做相应的累计计数操作就可以了。

领资料

为了更快地查找到单词的计数器，有两种优化思路，一种是我们上一讲学习的基于哈希的思想，直接将符号表映射到一个连续线性的数组空间，从而获得 O(1) 的访问时间复杂度；**另外一种思路就需要维护一个有序排列的符号表，JDK 中的 TreeMap 就是基于这种思路。**

试想，如果能够让符号表是有序排列的，我们查找的时候是不是就不用遍历每一个元素，而可以采用二分查找之类的手段了呢？当然也要尽量降低维护这个有序排列的数据结构所花费的代价。

那一种常见的用于实现有序集的数据结构就是红黑树，这也是 JDK 中 TreeMap 中 Tree 的意思。如果你有一定的 Java 开发经验，相信你一定会知道相比于 HashMap，**基于红黑树的 TreeMap 的一个显著特点就是其维护的键值对是有序排列的**。

如果你一听到红黑树这个词，就有点慌张，觉得这不是自己能驾驭的，今天这节课就来帮你打消这个顾虑。

很多人觉得红黑树很难理解，其实很大程度是因为无论是本科的教学还是网上流传广泛的资料，大部分只是描述了红黑树的一些规则和结论，这就让学习者往往只能死记硬背红黑树是“由红色黑色节点构成的一种近似平衡树”、“不能有连续的两个红色节点”之类的规则，而不知道为什么有这样的规则，当然会非常复杂，很难理解。

但是**如果我们追本溯源，从红黑树为什么被发明出来讲起，让你了解红黑树的本质，你就会发现它相当简单好理解了**。

而且相信大部分程序员应该不会有什么机会要手写红黑树，在正经面试中也绝对不会碰到手写红黑树这样的题目，所以我们掌握红黑树的本质和特性就足够了。当然，红黑树的部分实现细节，我们在最后也会讲到，如果你学有余力可以尝试自己实现一下。

好啦，我们开始今天的学习。

二分查找树

要想轻松理解红黑树，我们需要先来简单复习一下二分查找树和平衡二分查找树的概念，因为红黑树就是一种近似平衡的二分查找树实现。

我们知道，在一个线性的链表里，如果要查找某个特定节点，唯一能做的事情就是从头到尾遍历节点，这带来了平均为 O(N) 的查找时间复杂度。但是如果数据存储在有序的二分查找树上，情况就大有不同。

二分查找树的二分是指什么呢？

举个非常简单的例子，你在便利店购物，有一个商品忘记扫码触发门禁警报，怎么从一堆商品里迅速定位呢？一个一个扫显然很慢，聪明的方法是将商品先分为两堆过门禁，找到触发的那堆再二分，继续下去，就能迅速找到啦。本质就在于**二分每次可以排除掉一半的查询范围**。

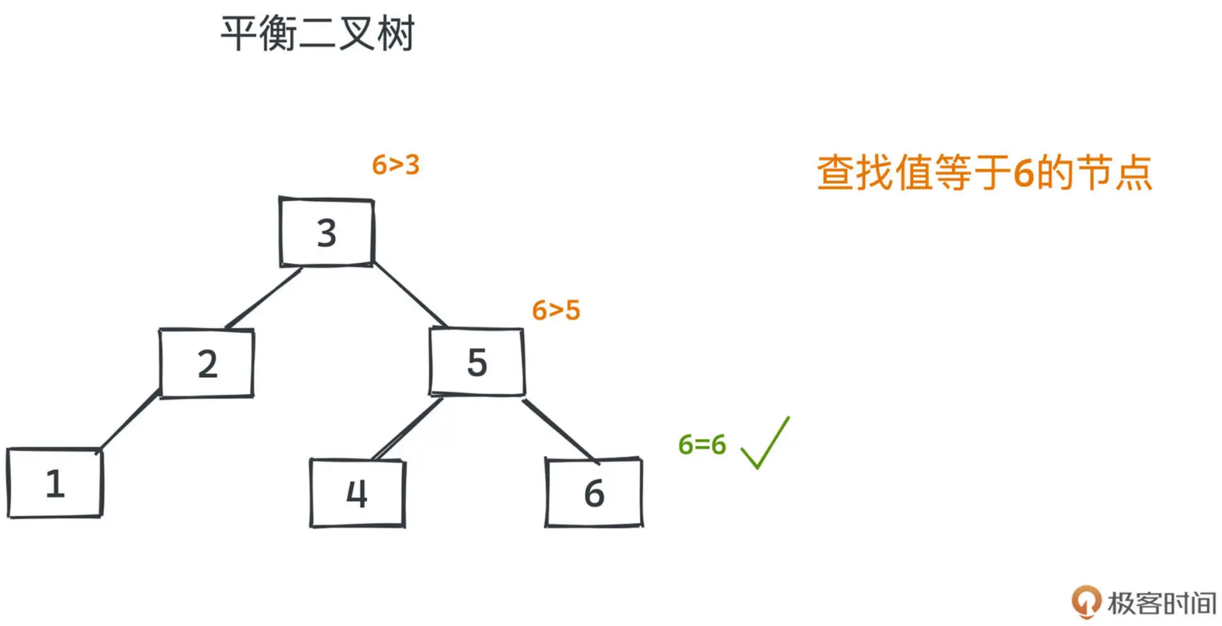
二分查找也是非常常见的算法。

比如在有序排列的数组中，因为数组访问任意下标元素的时间复杂度都是 O(1)，我们就可以通过二分查找法，在 O(logN) 的时间复杂度里，快速定位任意元素的位置。我们之后有一讲会详细学习数组上的二分搜索算法。

而二分查找树，则是另外一种可以用来实现二分搜索的数据结构。具体来说，我们在一棵普通的二叉树上放置需要存储的符号表，并保证所有节点和其左右子节点满足这样的关系：每个节点的左节点要不为空，要不为比当前节点小的元素（符号）；每个节点的右节点要不为空，要不为比当前节点大的元素（符号）。

在这样的约束下，我们的查找只需要从根节点出发，比较当前节点和目标元素之间的大小，要么往左走，要么往右走；**这是因为比当前节点大的元素一定在当前节点右边，反之则在当前节点左边，所以我们每次比较总可以排除左右子树中的一颗**。

反复进行这样的搜索过程，直到当前节点已经是叶子结点，或者当前结点和目标元素相等为止。需要比较的次数最多为树的最大高度 h，因此整个搜索过程的时间复杂度就是 O(h)，显然，O(h) 在大部分时候是一个比 O(n) 小得多的数。

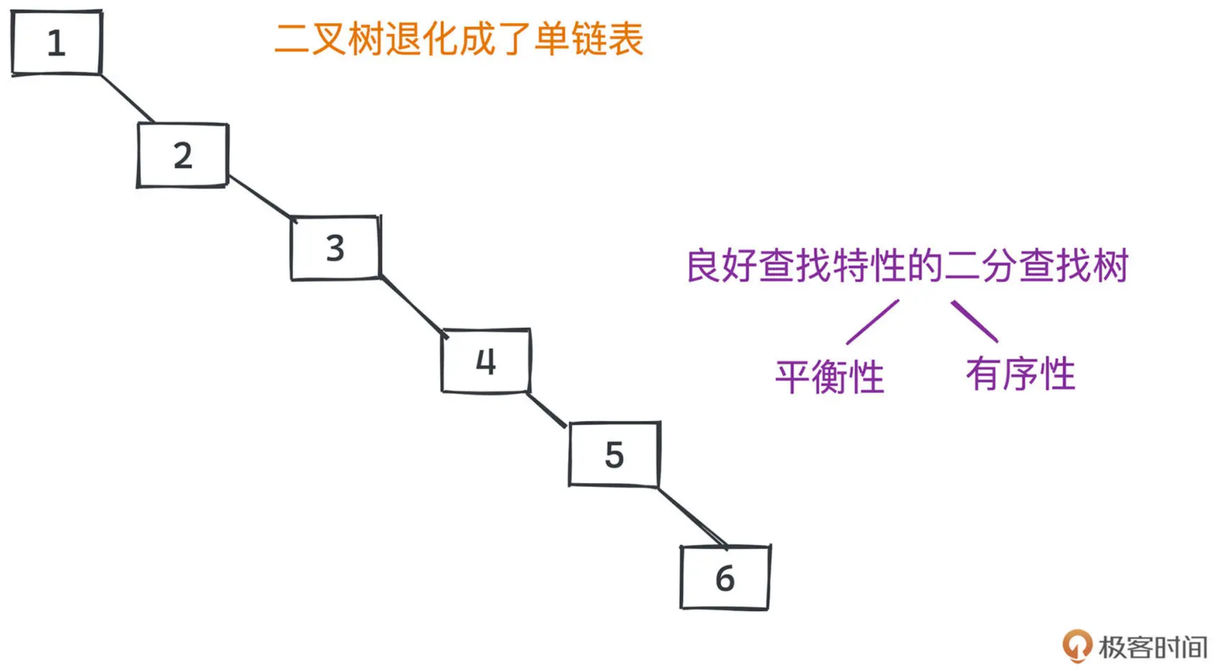


比如要查找值等于 6 的结点，在原始的链表中我们可能需要遍历完整个链表，需要 6 次，而在二分搜索树中，我们只需要沿着根节点一路往右子节点遍历，一共比较 3 次即可。

但是这样的二叉树在极端的情况下也会退化成链表。

平衡二分查找树

比如下图也是一个满足约束的二分查找树，但所有的结点都在树的一侧排列。在这样带有极度偏向性的树中，我们查找节点的效率其实和链表没有什么区别，反而还用了更多的空间。



所以理想中，**如果要实现具备良好查找特性的 OrderedMap，我们需要同时保证树的有序性和平衡性**，这里平衡性指的是，树上每个节点的左右子树的高度差要尽量小，最好不要超过一。比如前面的图就是一个很好的平衡二分查找树，这里的图就是一个退化成单链表的情况。

对于一个平衡的有 N 个元素的二分查找树，其高度可以近似认为是 logN，所以查找的时间复杂度就是 O(logN)。 这意味着一张有 10000 个元素的符号表，我们想要查询出任意一个元素，最多也只需要进行大约 13 次左右的比较即可。这显然是一个非常令人满意的结果。

哦，为什么高度是 O(logN) 呢？我们近似计算一下。

以满二叉树为例，也就是一颗除了叶子结点之外的结点都有两个子节点的二叉树。相信你很容易发现每一层能承载的容量范围都是上一层的一倍，那么，**最多在第 logN 层，只把这一层的节点加起来，就足够承载需要存储的所有 N 个元素了**。

而平衡树相比于满二叉树的最大高度显然是偏差有限的，高度应该也就是 O(logN) 这个数量级了。

那如何才能兼顾有序性和平衡性呢？这就是我们今天的主角红黑树出场的时候了。

红黑树

没错，红黑树就是这样一种兼顾了有序性、平衡性的自平衡二叉树的实现。因为它查找高效，成为许多语言实现内置 ordered_map 的首选，另外在 Nginx 的 Timer 管理、Linux 虚拟内存管理等场景下，红黑树也都承担着重要的角色。

其实之前还有人提出了一些不同的平衡二分搜索树的实现，比如 AVL-Tree、2-3-4 树等，但因为这些实现保持绝对平衡的代价比较高且往往实现复杂，并没有像红黑树这样遍布于各大需要有序键值对存储的场景。

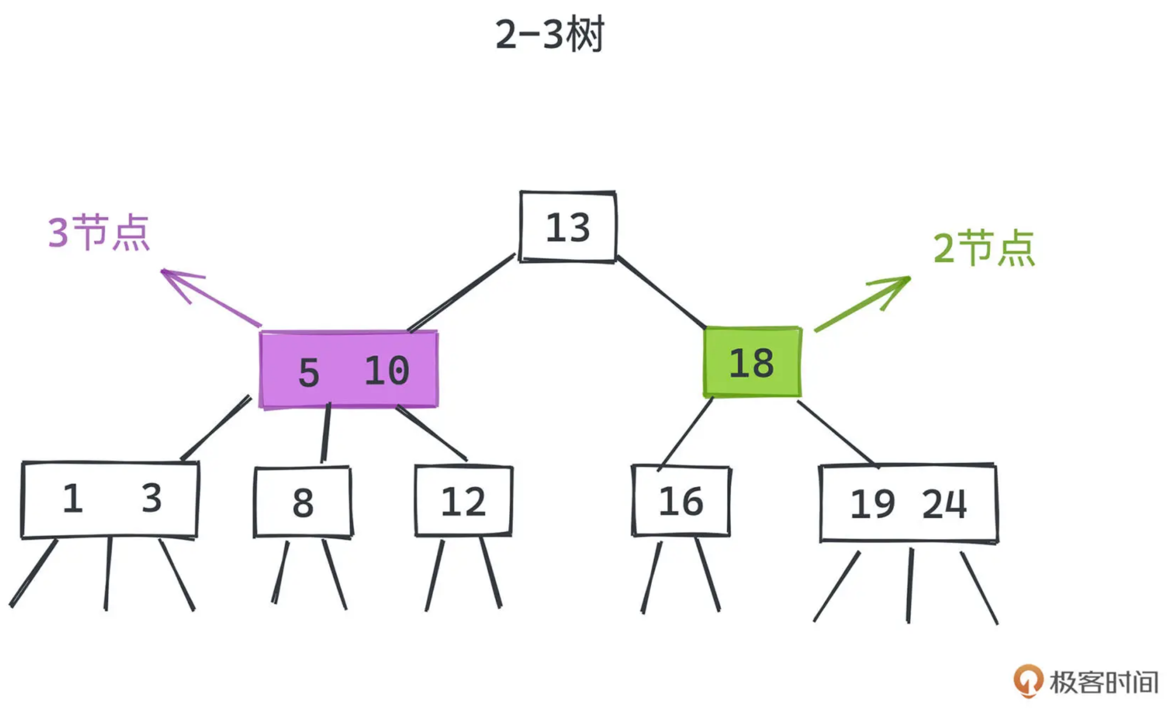
那么红黑树到底是如何实现的呢？

要更好地理解红黑树的设计，首先你要理解红黑树的由来——红黑树本质上是对“2-3 树”的一种实现。

那就让我们先一起来学习一下“2-3 树”的实现，相信你学完一定会恍然大悟，红黑树这么奇怪的红色黑色的节点设计原来是这样来的。

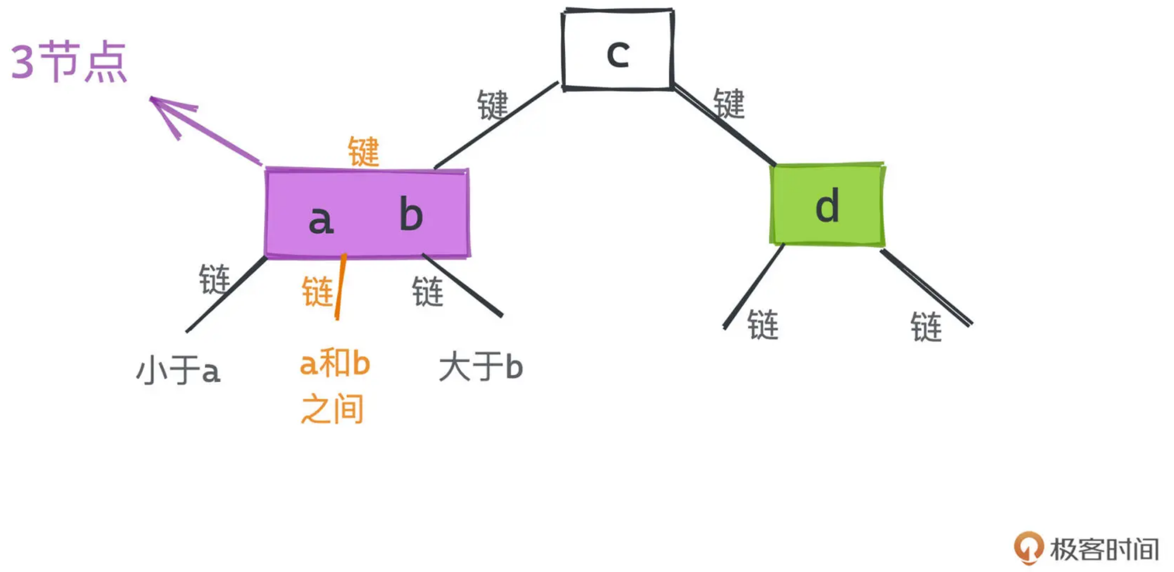
2-3 树

2-3 树，也是一种平衡查找树的实现，思想很简单，为了让树能更好地平衡自己，我们除了普通的 2 节点之外，还引入了一种 3 节点，这让我们在平衡树高度的时候增大了很大的灵活性。看一个典型的 2-3 树例子。



在 2-3 树中，2 节点，和普通二叉树的节点其实没有什么太大的区别，有一个键和两条链，分别链向左子树和右边子树。

而 3 节点，则在 2 节点的基础上增加了一个键，构成了一个有两个键和三条链的结构。下图是 3 节点的示意图，左链链向了小于 a 的节点，右链链向了大于 b 的节点，中间的区域则是 a 和 b 之间的节点。



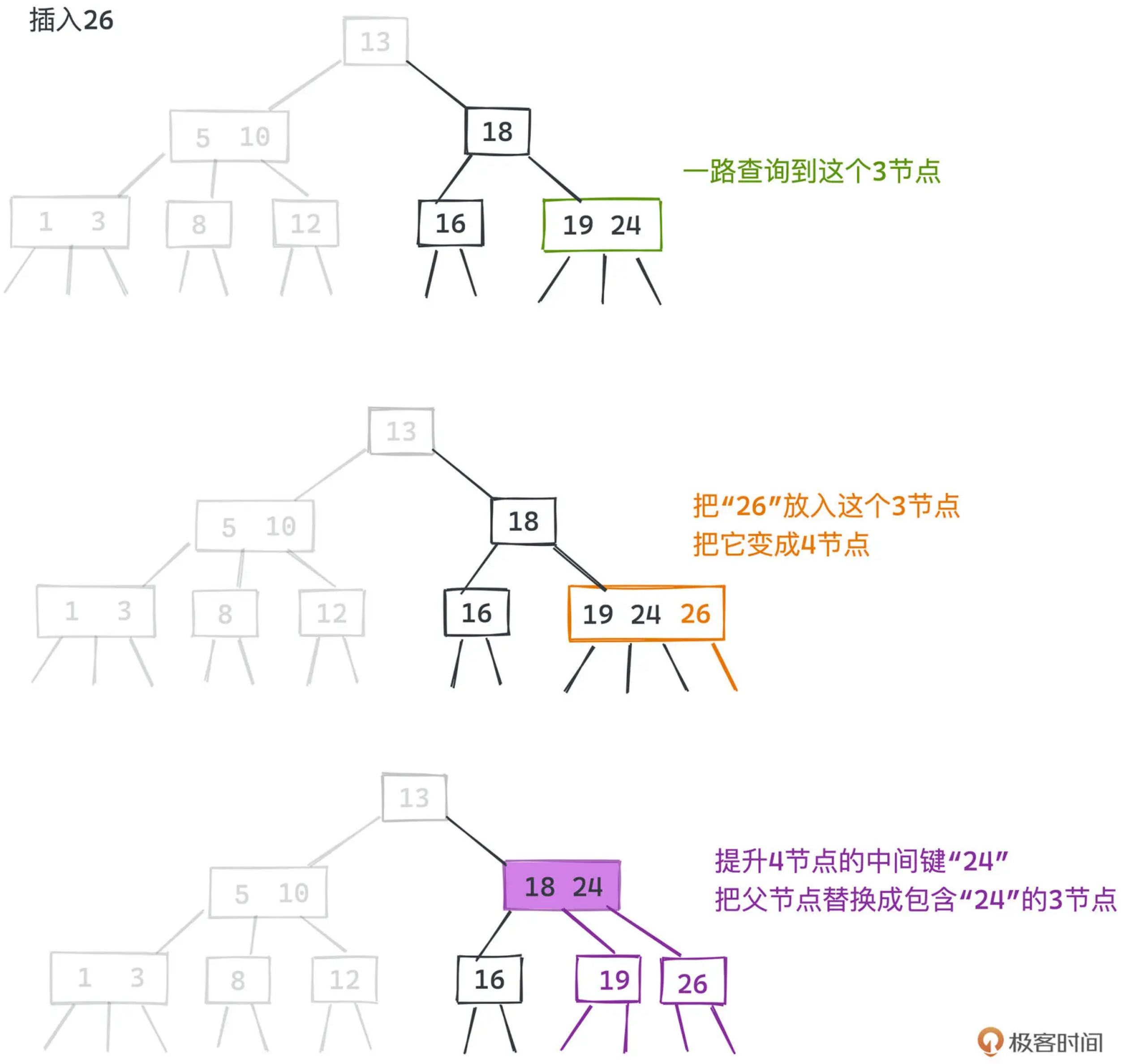
在 2-3 树中搜索的过程和二叉树并没有太多的区别，只是遇到 3 节点的时候，多判断一次是否介于 a、b 之间即可。

设计有了， 我们看插入新元素的时候会发生什么。

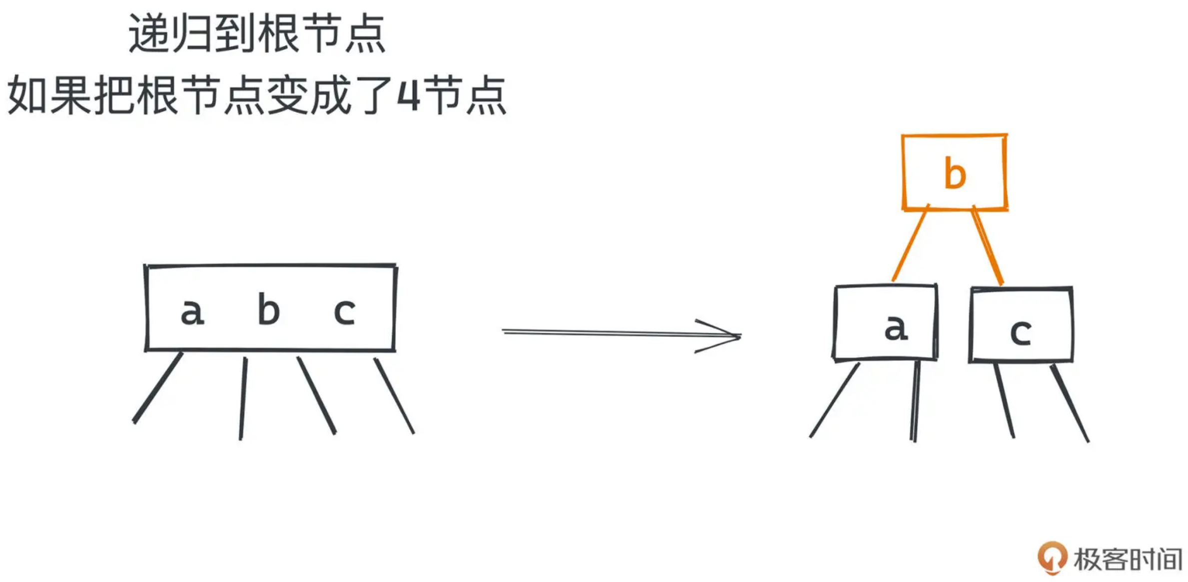
在插入过程中，当我们查找到了某个叶子结点发现并不存在该键时，如果遇到了 2 节点，非常好办，直接加个键将该节点升级为 3 节点即可。比较麻烦的是遇到了 3 节点，因为我们已经不能再在该节点中直接多加一个键创造一个 4 节点了，怎么办呢？

其实办法也不难想，我们把当前的 4 节点多出的键向上转移。看图理解，比如要对下图中的 2-3 树插入 26 节点，那首先会沿着根节点一路查询到 “19 24” 子结点，发现该节点为一个 3 节点。

那么我们首先**将 26 放入该子节点，使之成为一个 4 节点，然后将 4 节点的中间键也就是 24，提升到上一层，将其父节点替换成一个包含 24 的 3 节点**。



如果原父节点也是一个 3 节点的话，我们就递归进行同样的操作直至根节点。最后，如果根节点也是一个 3 节点，我们就将根节点的中键提升到第一层，然后左右链分别链向原来根节点的左键和右键。以下图为例，b 键就被独立地提升为新的根节点，左右节点指向 a 和 c，而原 4 节点的中间两个链也分别成为 a 的右链和 c 的左链。



这样的操作可以保证整个 2-3 Tree 是一个真正意义上的平衡树。但是，因为它的实现引入了两种异构的节点，导致代码写起来相当复杂，并没有被广泛使用。

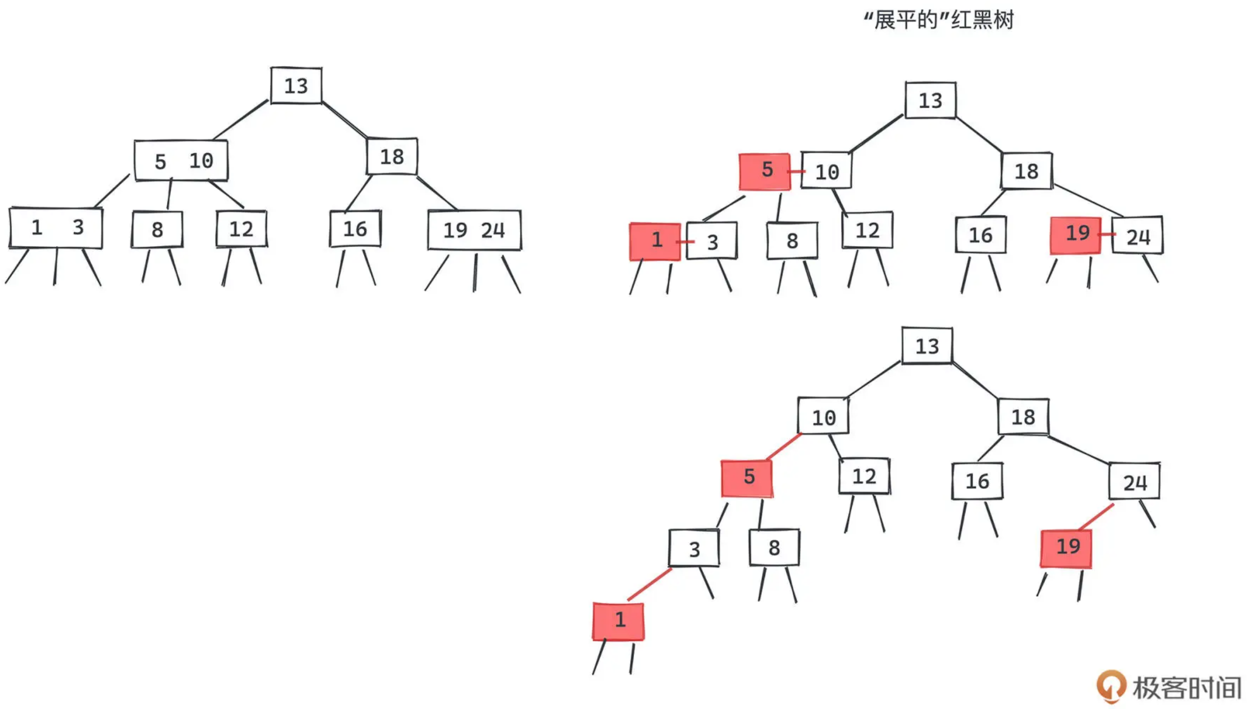
而红黑树，**正是采用标准的二叉查找树节点附着上额外的颜色信息来表示 2-3 树的实现，每一个红色节点都和它的父亲节点一起，构成了一个 3 节点的模拟，这就是红黑树设计的本质。**

红黑树

所以，我们再把红黑树的定义拿出来，红黑树是一个满足下述几个约束且所有节点要么为红色要么为黑色的二分有序查找树：

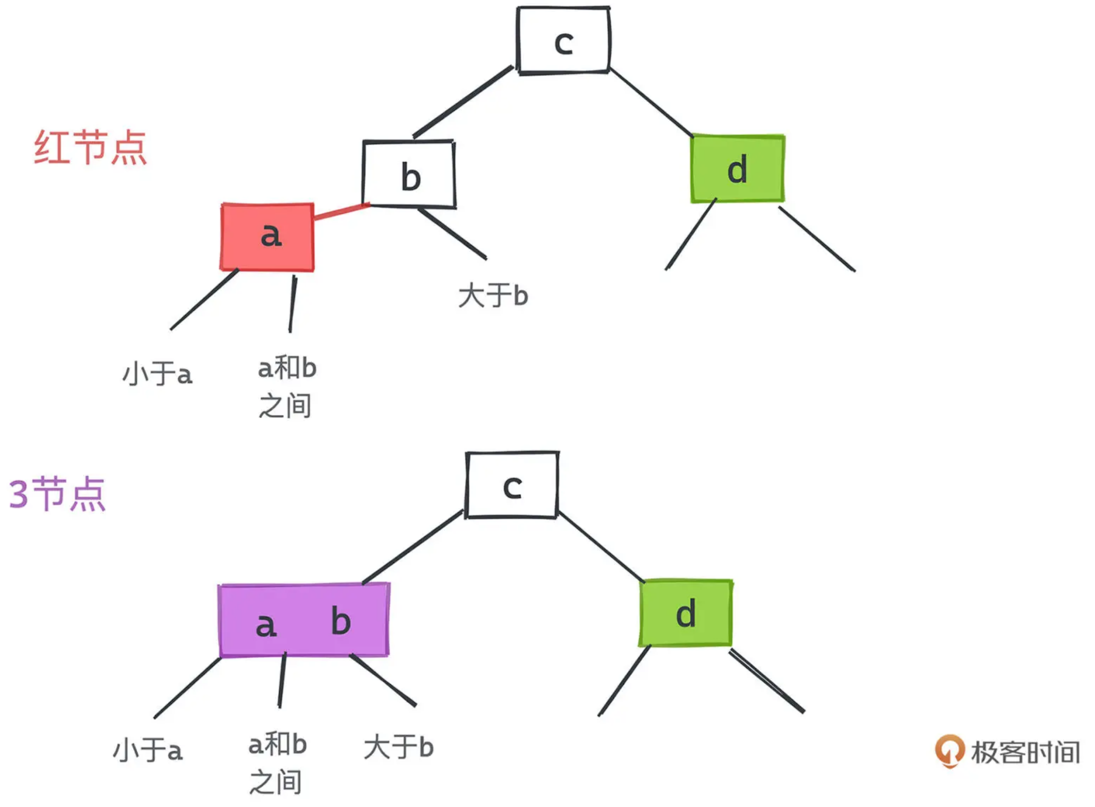
1. 根节点为黑色
2. 相邻的节点不能同时为红色
3. 每个节点从自身到各个能到达的叶子结点的所有路径中的黑色节点数量相等
4. 红节点只能作为左子节点存在（这是左偏红黑树特有的要求，我们以左偏红黑树为例讲解）

所有这些约束，都是为了保证每一颗红黑树和 2-3 Tree 是一一对应的，相信你看下面这颗“展平”的红黑树就能理解我在说什么了。



我们一起顺一下。

一个 3 节点有两个键、三条链，那我们完全可以把一个以红节点为左子节点的黑节点和子节点一起看成一个 3 节点。在下图中，上下两个图其实就可以认为是等价的。



我们再来看红黑树的 3 个普遍约束，你会发现很好理解：

1. 因为红色节点只是 3 节点的一部分，那对应到红黑树上，显然不会出现两个连续的红色结点；
2. 2-3 树上，每个节点到叶子节点的数量一定是一样的，且每个节点对应到红黑树上一定包含且只包含一个黑色节点，所以红黑树每个节点到叶子结点路径中的黑色节点数量也必然是一样的。

唯一不一样的就是“根节点为黑色”。事实上，如果只是为了让红黑树保持平衡，我们完全可以抛弃这条规则。因为在 2-3 树中，我们也完全是可以用 3 节点作为根节点的。

对应到红黑树中，当根节点为红色，插入新节点后很可能为了使根节点到每条路径上的黑色节点数量相等，进行变色和旋转操作，最终根节点还是会变成黑色；既然如此，我们何不直接约束根节点必须调整成黑色，方便进行插入操作呢。

这样，我们就可以将每一个红黑树都映射成一个 2-3 树，也因此就获得了 2-3 树高效的平衡插入的能力，并保留了二叉树查找的简洁性。之后在理解红黑树的时候，如果你能时刻展平成 2-3 树看待，一定会觉得，哦，红黑树的实现其实也没有想象中的那么困难。

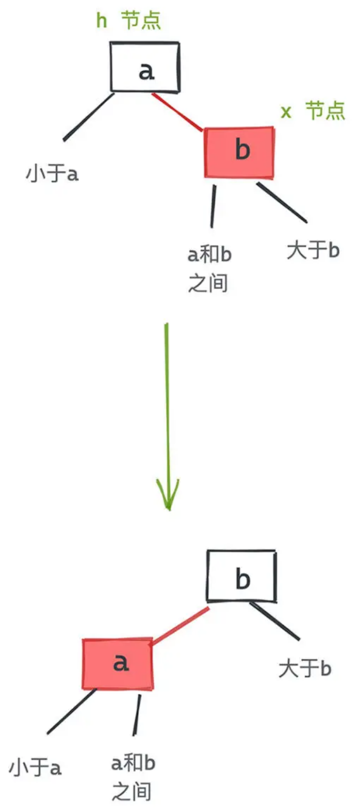
最后，我们来看一些红黑树的基本操作，帮助你更好地理解红黑树和 2-3 树之间的关系。

旋转操作的实现

红黑树的所有实现细节，其实也都是围绕着 2-3 树的 2、3 节点的诞生和转移展开的，我们就以“插入方法”的实现来具体讨论（仍以左偏红黑树为例）。

红黑树中最基本的自平衡操作就是“旋转”，分为“左旋”和“右旋”两种。这两种操作主要用于处理在插入和删除时产生的右偏红节点或者连续的两个红色节点，通过调整红节点的位置，我们可以修复这些不满足约束的情况。

看“左旋”和“右旋”的具体操作。**以左旋为例，本质上就是将某个 3 节点从以较小的键为根转移成较大的键为根**，也就是从 a 为根转到 b 为根，当然同时需要把介于 a 和 b 之间的节点挂到 a 的右节点下。这样得到的新树就是以 b 为根结点的结构，并且在整个过程中，树的平衡性和有序性都没有被破坏，而原来不符合约束的右偏红节点已经被转移成“正确”的左偏红节点。



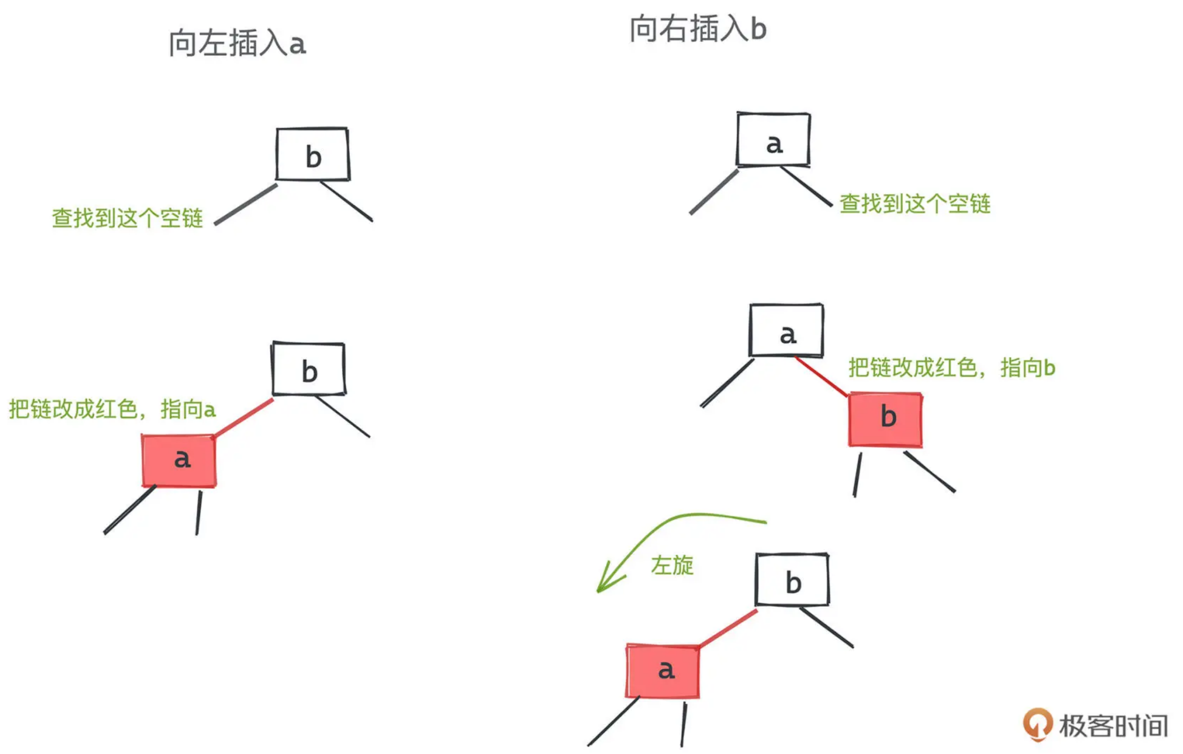

```
Node rotateLeft(Node h)
{
  Node x = h.right;
  h.right = x.left;
  x.left = h;
  x.color = h.color;
  h.color = RED;
  x.N = h.N;
  h.N = 1 + size(h.left)
          + size(h.right);
  return x;
}
```

现在，我们根据插入时是在“2 节点”还是“3 节点”分开讨论，看看旋转操作具体是如何用于保证约束正确的。

2 节点插入

首先我们来看针对“2 节点”的插入，对应到红黑树的语境中，也就是针对普通黑色节点的插入。那显然只有两种情况，要么插入在左边，要么插入在右边。

如果插入在左边，非常简单，我们可以直接将 2 节点提升为 3 节点，由于新增的是左侧的红节点，完全不会破坏树的平衡性。对应到红黑树上，就是简单地将新节点放到查找到的最后一个节点的左边。

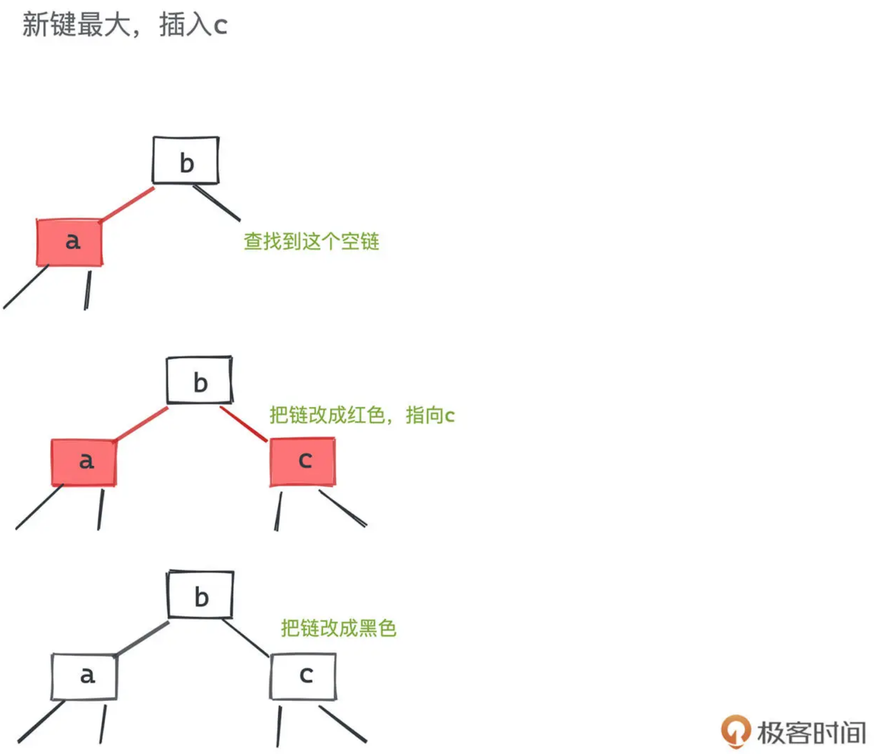


如果插入在右边，其实是一样的，我们也将 2 节点提升成一个不符合“左偏”规则的 3 节点，然后进行一次左旋转即可。由于插入的也是红节点，并不影响树的平衡性。

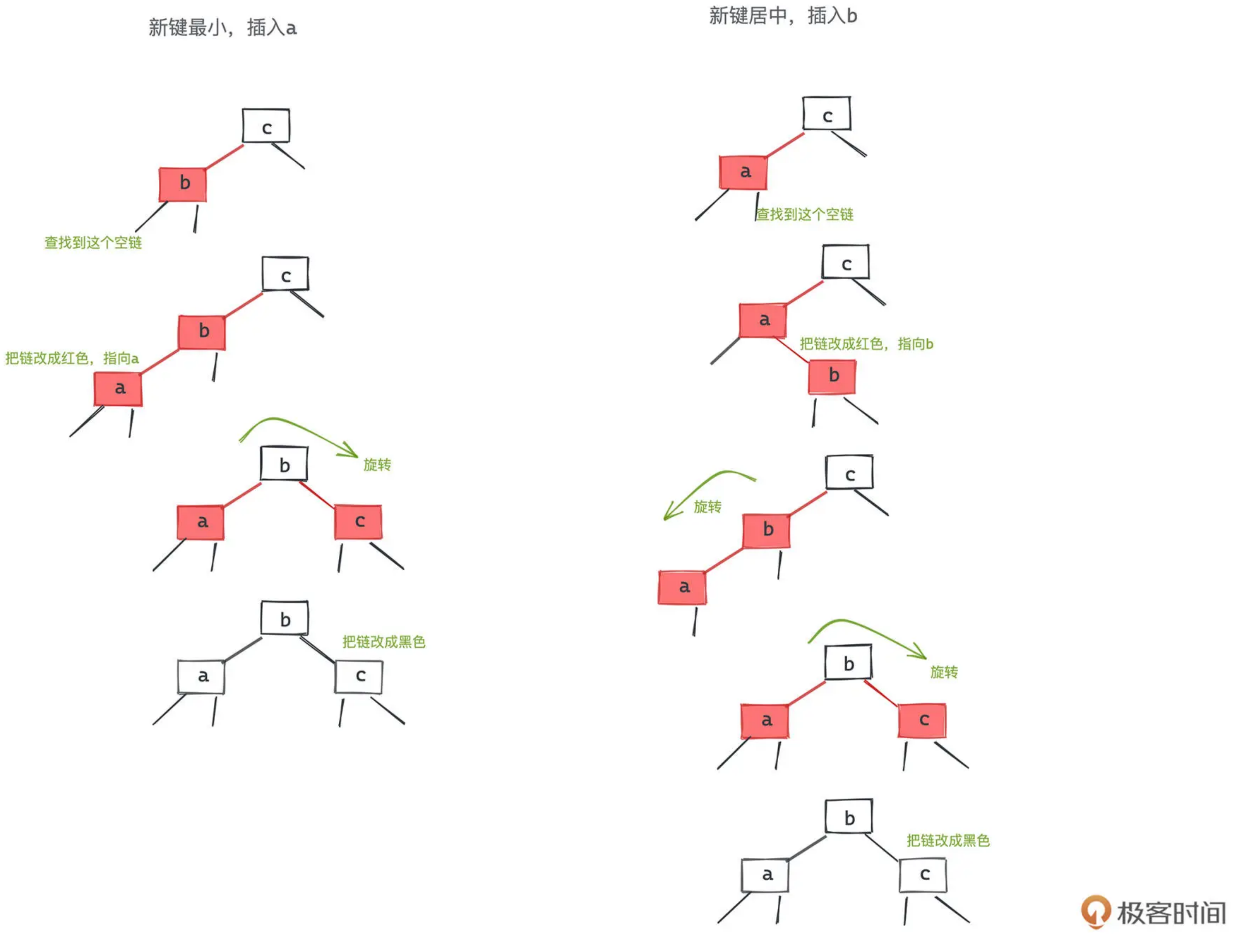
3 节点插入

再看插入“3 节点”，情况和操作都会复杂一些，我们根据插入的结点在 3 节点中左键的左侧、右键的右侧和两者之间分开讨论（还是左偏红黑树）。

最简单的情况就是插入在右键的右侧。和 3 节点分解的方式一样，我们只需要把中间的结点提升到上一层，并左右节点变成黑色即可。



而另外两种情况单独看这一个节点的变化，也并不复杂，只需要插入后进行一到两次旋转操作即可。

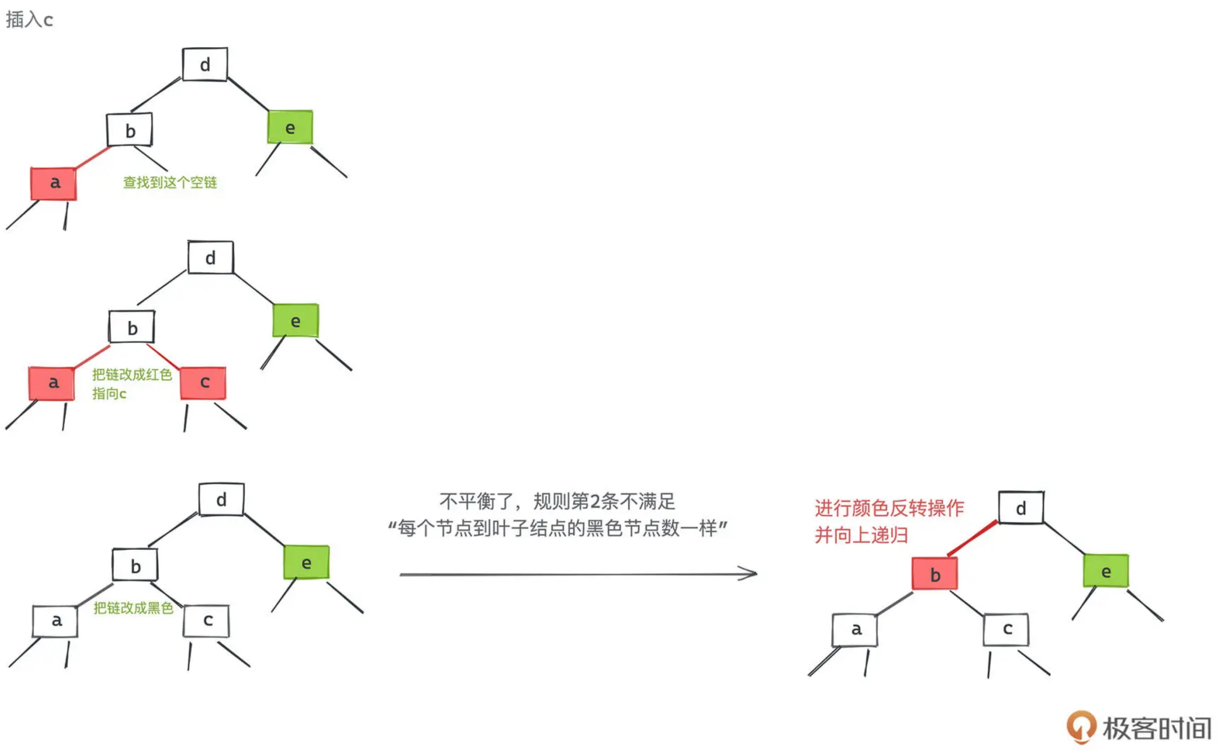


但这样是不是就完成了所有操作呢？ 并不是。

还有一个很大的问题我们没有处理，就是对 3 节点的操作中，我们虽然保证了“插入操作”对当前子树的“平衡性”没有被破坏，但由于将红色节点变成了黑色，就有可能导致当前子树的黑色节点高度比其他子树高了。

所以我们还需要进行一种叫做“颜色反转”的操作。

每次插入时，**最后一步，除了将 3 节点的左右节点都变成黑色，同时要将 3 节点的中间键变成红色**，这样当前子树到各个子节点路径中的黑色节点数量就不会有变化啦。



当然这个操作是需要递归进行的。因为父节点如果变成红色，也同样可能造成右偏红节点或者连续红节点这样不符合约束的情况，这其实等价于在父节点的父节点下插入了一个新的红节点。我们用类似的逻辑自下而上递归即可，递归的终点就是，遇到根节点我们将其拆分成 3 个“2 节点”，或者遇到某个 2 节点我们将其升级为“3 节点”时，我们就可以结束递归。

讨论好这些 case，我们就可以在整个插入节点的过程中**保证不破坏“有序性”和“平衡性”**了。

删除的操作和插入类似，感兴趣的话，你可以课后自己通过纸笔模拟一下。整个过程确实比较复杂，许多细节过一段时间可能也会有所遗忘，不过只要你理解了红黑树的本质，是在二叉树基础上对“2-3 Tree”的实现，一定能迅速回忆起红黑树的约束条件。到这里，无论是准备面试，还是帮助你了解许多用到红黑树的中间件源码来说，都已经绰绰有余了。

操作红黑树的复杂度

最后我们来稍微计算一下红黑树上检索数据的复杂度。

前面我们提到，二叉搜索树检索键的最差时间复杂度取决于树的最大高度，是 O(logN)，那求红黑树上检索数据的时间复杂度就等同于求红黑树的最大高度了。

我们将一个红黑树展平，并对应到 2-3 树。

2-3 树是一颗平衡的树，由于每个节点只可能比二叉树多出一个键，在相同的高度下只会承载更多的元素，所以其高度不可能高于二叉平衡树的高度 logN。

而红黑树只是把 2-3 树的每个“3- 节点”拆成了一黑一红两个节点，其高度不可能比 2-3 树的最大高度翻倍还多；当然你也可以从“红色节点不能相邻”这一约束得出类似的结论，毕竟本质这两者是等价的。

所以，满足这样约束的红黑树的最大高度最多也就是 2*logN，因此可以得到良好的 O(logN) 的查询复杂度。插入、删除复杂度显然也只和高度有关，我们最多需要进行常数倍于树的最大深度次旋转和颜色反转操作，所以复杂度也是 O(logN)。

统计单词数量

好了，有了红黑树这样一个可用于动态、高效查找 / 删除 / 插入数据的数据结构，维护一个符号表也当然就很简单啦。如果依旧让你统计某文档中不同的单词出现的数量，我们就可以维护一颗以不同单词为键，出现次数为值的红黑树，在比较树的有序关系时只比较键的关系即可。

这样，每次遇到一个新的单词就在红黑树中查找相应的键是否存在，如果有存在就将对应节点的出现次数自增，如果没有存在就插入一个新的节点，让其值为 1。

写成 Java 代码，你会发现和上一讲 HashMap 也没有有什么区别，只是将 HashMap 换成了 TreeMap 而已，底层机制就我们刚刚梳理的一样。

复制代码

```java
import java.util.TreeMap;
import java.util.Map;
public class Test {
    public static void main(String[] args) {
        Map<String, Integer> map = new TreeMap<>();
        String doc = "aaa bbb ccc aaa bbb ccc ccc bbb ccc ddd";
        String[] words = doc.split(" ");
        for (String s : words) {
            if (!map.containsKey(s)) {
                map.put(s, 1);
            } else {
                map.put(s, map.get(s) + 1);
            }
        }
        System.out.println(map);
    }
}
```

假设有 N 个单词，则红黑树最大高度为 logN，所以单词查询的最大时间复杂度为 O(logN)，这样的查询一共会进行 N 次，所以整体的时间复杂度为 O(NlogN)，从时间效率上来说确实是低于 HashMap 的。

总结

红黑树的查询、插入、删除的时间复杂度都非常良好且稳定，广泛运用于各种中间件里，还是非常值得掌握的。

当然红黑树是一个比较复杂的数据结构，如果直接从定义和几条绕口令一样的约束入手，很不好学。不过只要掌握它的本质，其实很简单。只要搞清楚，**红黑树本质是 2-3 树在二叉树上的一种模拟，通过旋转操作完成 2-3 节点的合并和分裂，从而在不改变二叉树节点结构的前提下，保证二叉树的有序性和平衡性。**

相信你理解了这一层关系，就可以对红黑树有一个比较不错的掌握了；相比于只是死记硬背红黑树的各种特性和复杂约束，追本溯源显然是一个更好的选择，这在你学习技术知识乃至其他领域，都是值得推荐的学习方式。

课后作业

那什么时候我们该用 TreeMap 什么时候该用 HashMap 呢？欢迎你留言与我一起讨论哦！

拓展阅读

给感兴趣的同学推荐一下我觉得最好的学习材料《Algorithms》，作者是左偏红黑树的发明者。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | HashMap：一个优秀的散列表是怎么来的？

下一篇 07 | 堆：如何实现一个高效的优先队列？

精选留言 (5)

写留言

Paul Shan
2021-12-23

红黑树每次看的时候都好像明白了，过一段时间就会忘记旋转细节。但是，红黑树确实是工程实现的典范，为了让完全二叉树容纳各种数量的节点，引入了2，3树的概念，也就是保持树的形状为完全二叉树，让树的节点容纳一个或者两个元素来承载变化。有了这个灵活性，平衡二叉树的实现就方便很多，而且也高效不少（半数插入不用调整树的结构）。增加节点的颜色来代替异构节点简化了实现。引入了根节点为黑色和红色节点只出现在...

展开

共 1 条评论 2

qinsi
2021-12-23

左偏红黑树和算法4作者的亲自讲解

https://www.coursera.org/lecture/algorithms-part1/red-black-bsts-GZe13

这是白猫

2021-12-23

HashMap 和 TreeMap 都是使用红黑树实现的，区别是HashMap多了外面的基于hashCode的有序链表，因此在重复数据较多时使用Treemap，反之大量不重复数据时使用 HashMap，是不是这样呢老师

2021-12-23

红黑看过几个版本 一个是算法4 一个是邓公的 关于旋转操作我看过邓公的那个 connect34 方法 我惊了我根本没想到可以这样通过打散然后组合的方式

blentle

2021-12-23

avl 实现更简单，比红黑树性能并没有差多少，为什么应用场景没有红黑树那么广呢

共 2 条评论 >